# Provably Optimal Parallel Transport Sweeps on Regular Grids

M. P. Adams, M. L. Adams, W. D. Hawkins, T. Smith, L. Rauchwerger, N. M. Amato, T. S. Bailey, R. D. Falgout

February 14, 2013

International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering
Sun Valley, ID, United States
May 5, 2013 through May 9, 2013

*International Conference on Mathematics and Computational Methods Applied to Nuclear Science & Engineering (M&C 2013)*
Sun Valley, Idaho, USA, May 5-9, 2013, on CD-ROM, American Nuclear Society, LaGrange Park, IL (2013)

# PROVABLY OPTIMAL PARALLEL TRANSPORT SWEEPS ON REGULAR GRIDS

**Michael P. Adams[1], Marvin L. Adams[1], W. Daryl Hawkins[1], Timmie Smith[2], Lawrence Rauchwerger[2], and Nancy M. Amato[2]**
[1]Dept. of Nuclear Engineering; [2]Dept. of Computer Science and Computer Engineering
Texas A&M University, 3133 TAMU, College Station, TX 77843-3133
dhawkins@tamu.edu; timmie@tamu.edu;

**Teresa S. Bailey and Rob D. Falgout**
Lawrence Livermore National Laboratory
bailey42@llnl.gov; rfalgout@llnl.gov

## ABSTRACT

We have found a set of provably optimal algorithms for executing full-domain discrete-ordinate transport sweeps on regular grids in 3D Cartesian geometry. We describe these algorithms and sketch a "proof" that they will always execute the full eight-octant sweep in the minimum possible number of stages for a given $P_x \times P_y \times P_z$ partitioning. A stage includes each processor choosing a task to execute, if at least one is available, and communicating its result to downstream neighbors. A task is an aggregation of cells, directions, and energy groups. Our computational results demonstrate that our optimal scheduling algorithms do execute sweeps in the minimum possible stage count, and further they agree reasonably well with our performance model. An older version of our PDT transport code achieves almost 80% parallel efficiency on 131,072 cores, on a weak-scaling problem with only one energy group, 80 directions, and 4096 cells/core. A newer version is less efficient at present—we are still improving its implementation—but still achieves almost 60% parallel efficiency on 393,216 cores. These results conclusively demonstrate that sweeps can perform with high efficiency on core counts approaching one million.

*Key Words*: List of at most five key words

## 1. INTRODUCTION

The full-domain "sweep," in which all angular fluxes in a problem are calculated given previous-iterate values only for the volumetric source, forms the foundation for many iterative methods that have desirable properties[1]. One such property is that iteration counts do not change with mesh refinement and thus do not grow as resolution is increased in a given physical problem—an important consideration for the high-resolution transport problems that are the focus of massively parallel computing. A transport sweep is defined as the calculation of all angular fluxes in the problem given some guess or iterate for the total source. That is, a sweep calculates $\psi_{m,g}^{(l+1/2)}$ via the numerical solution of:

$$\vec{\Omega}_m \cdot \vec{\nabla}\psi_{m,g}^{(\ell+1/2)} + \sigma_{t,g}\psi_{m,g}^{(\ell+1/2)} = q_{tot,m,g}^{(\ell)} \quad , \tag{1}$$

where $q_{tot,m,g}^{(\ell)}$ includes the collisional source evaluated using fluxes from a previous iterate or guess (denoted by superscript $\ell$). We emphasize that this is a complete boundary-to-boundary sweep of all directions, respecting all upstream/downstream dependencies, with no iteration on interface angular fluxes. For the analysis in this paper we assume that the incident angular flux is specified on each incoming surface of the problem, with no reflecting or periodic boundaries. We expect to address reflecting and periodic boundary conditions in a later communication.

The parallel execution of a sweep is complicated by the dependency of a given cell on its upstream neighbors. A simple task dependence graph (TDG) for a single quadrature direction in a 2D example (Fig.1a and b) illustrates the issue: tasks at a given level of the graph cannot be executed until some tasks finish on the previous level. This issue has led to a common perception that parallel sweep execution will not be efficient beyond a few thousand parallel processes[2,3] and has led researchers to seek iterative methods that do not use full-domain sweeps[2,3]. These methods offer the possibility of easier scaling to high processor counts—for a single iteration's calculation—but typically the iteration count will increase as each processor's portion of the problem domain decreases.

In this paper we focus on discrete-ordinates transport sweeps and describe new parallel sweep algorithms. We demonstrate via theory, models, and computational results that our new optimal sweep algorithms enable efficient parallel sweeps out to O($10^6$) processors, even with modest problem sizes (a few hundred thousand cell-energy-direction elements per processor). In this paper we assume a regular orthogonal grid of $N_x \times N_y \times N_z$ spatial cells, $G$ energy groups, and $M$ quadrature directions in each of the eight octants. Irregular grids, such as AMR grids or non-orthogonal grids, introduce complications that we will not address in this paper.

The KBA algorithm devised by Koch, Baker, and Alcouffe [4] is probably the best known parallel sweep algorithm. KBA *partitions* the problem by assigning a column of cells to each processor, indicated by the four diagonal task groupings in Fig.1c. KBA parallelizes over planes perpendicular to the sweep direction—over the breadth of the TDG. Early and late in a single-direction sweep, some processors are idle, as in stages 1-3 and 9-11 in Fig.1. In this example, parallel efficiency for an *isolated* single-direction sweep could be no better than $8/11 = 0.73$. KBA is much better, because when a processor finishes its tasks for the first direction it begins its tasks for the next direction in the octant-pair that has the same sweep ordering. That is, each processor begins a new TDG as soon as it completes its work on the previous TDG, until all directions in the octant-pair finish. This is equivalent to concatenating all

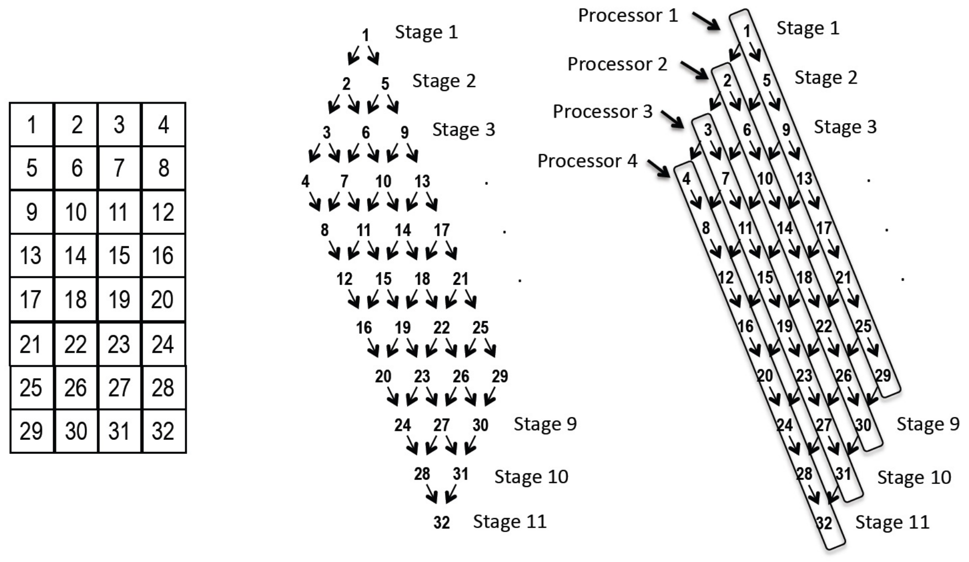

**Figure 1.** (a) Example 2D problem with $8 \times 24$ spatial grid aggregated into $4 \times 8$ cellsets. (b) Example TDG for a single direction's sweep. (c) KBA partitioning, with a columns of cellsets assigned to each of four processors. Tasks on a given level of the graph can be executed in parallel.

of an octant-pair's TDGs into a single much longer TDG. This effectively lengthens the "pipe" and increases efficiency. If there were $2M$ directions in the octant pair, then the pipe length is $2M \times 8$ in this example, and the efficiency would be $(2M \times 8)/(3 + 2M \times 8)$ if communication times were negligible.

A sweep algorithm is defined by its *partitioning* (dividing the domain among processors), *aggregation* (grouping cells, directions, and energy groups into "tasks"), and *scheduling* (choosing which task to execute if more than one is available). The work presented here builds upon the foundation given by KBA and on the work of Bailey and Falgout, who in 2009 theoretically and computationally evaluated the performance of three sweep algorithms including KBA[5].

In this paper we assume that the spatial grid is partitioned into a $P_x \times P_y \times P_z$ process grid, with $P =$ number of processes $= P_x P_y P_z$. The work to be performed in the sweep is to calculate the angular intensity for each of the $8M$ directions in each of the $G$ energy groups in each of the $N_x N_y N_z$ spatial cells, for a total of $8MGN_xN_yN_z$ fine-grained work units. The finest-grained work unit is calculation of a single direction and energy group's unknowns in a single cell; thus, we describe the sweeps that we analyze here as use "cell-based." Methods based on solutions along characteristics permit finer granularity of the computation; in particular, "face-based" sweeps are possible, and with long-characteristic methods “track-based” sweeps are possible. Face-based and track-based sweeps offer advantages over cell-based sweeps in terms of potential parallel efficiency. We expect to address this in a future communication, but in this paper we focus solely on cell-based sweeps.

We aggregate the fine-grained work units into coarser-grained tasks, with each task being the solution of the angular fluxes in $A_g$ groups, $A_m$ directions, and $A_x A_y A_z$ spatial cells. (The $A$ values are “aggregation factors.”) We set $A_x = N_x/P_x$ and $A_y = N_y/P_y$, which means a task includes an integral number of $x - y$ planes of cells in a processs subdomain. For our analysis we impose the restriction that the following ratios are all integers: $N_g = G/A_g$, $N_m = M/A_m$, $A_x$, $A_y$, $N_z/P_z$, and $N_k = N_z/(P_z A_z)$. It follows that each process owns $N_k$ cell-sets (each of which is $A_z$ planes of $A_x A_y$ cells), $8N_m$ direction-sets, and $N_g$ group-sets, for a total of $8N_m N_g N_k$ tasks.

The $A_m$ directions that are aggregated together are required to be within the same octant. The sweep for directions in a given octant must begin at one of the eight corners of the spatial domain and proceed to the opposite corner. Suppose for the moment that we had only one set of $A_m$ directions and one set of $A_g$ energy groups to calculate for each of the $P_x P_y P_z N_k$ cell-sets. Then the sweep would proceed from one corner to the other and would require $P_x + P_y + P_z N_k - 2$ stages for completion. (It takes $P_x$ stages for the sweep front to cross the $x$ dimension, then $P_y - 1$ more to cross the $y$ dimension, and $P_z N_k - 1$ more to cross the final dimension.) Two sets of directions in the same octant would require only $N_k$ additional stages, because the second set could be launched immediately after the first-corner process completed its $N_k$ stages for the first set. In general, of course, all eight octants are in play, with each starting at a unique corner of the spatial domain and ending at the opposite corner. If multiple direction-sets are launched at the same time, there will be collisions in which a process or set of processes will have multiple tasks available for execution. A *scheduling* algorithm is required for choosing which task to execute.

Scheduling algorithms are the main focus of this paper. We introduce a family of scheduling algorithms that execute the complete 8-octant sweep in the minimum possible number of steps (“stages”). We prove this mathematically for specific members of the family, and we present computational results that convincingly demonstrate this.

With such a scheduling algorithm in hand we know how many stages a sweep will require, for a given partitioning and aggregation, for a given problem. With stage count known, there is a possibility of predicting execution time via a performance model, and then using the model to choose partitioning and aggregation factors that minimize execution time for the given problem on the given number of processors. The result is what we call an “optimal sweep algorithm.” To recap, the ingredients of the optimal sweep algorithm are:

1. A sweep scheduling algorithm that executes in the minimum possible number of stages for a given problem with given partitioning and aggregation parameters;

2. A performance model that estimates execution time for a given problem as a function of stage count, machine parameters, partitioning, and aggregation;

3. An optimization algorithm that chooses the partitioning and aggregation parameters to minimize the model's estimate of execution time.

In the following section we discuss characteristics of parallel sweeps, including the processor idle time that must occur if sweep dependencies are enforced and the minimum possible number of stages required to complete a sweep for a given partitioning with given aggregation factors. We also develop and discuss simple performance models. The third section describes our optimal scheduling algorithms, which execute sweeps in the minimum possible number of stages. For one algorithm we prove optimality for three different kinds of partitioning, namely $P_z = 1$ (the KBA partitioning), $P_z = 2$ (which we call "hybrid"), and $P_z > 2$ ("volumetric"). This is the main contribution of this paper. In the fourth section we present our *optimal sweep algorithm*, which is made possible by our optimal scheduling algorithm. For optimal sweeps we automate the selection of partitioning and aggregation parameters that minimize execution time, as predicted by our performance model under the assumption that sweeps always complete in the minimum possible number of stages for a given set of parameters. Section 5 presents parallel transport results ranging from 8 cores to approximately 400,000 cores, with two different optimal-scheduling algorithms and one non-optimal algorithm. In all cases the optimal algorithms complete the sweeps in the minimum possible number of stages, and performance agrees reasonably well with the predictions of our performance model. We offer summary observations, concluding remarks, and suggestions for future work in the final section.

## 2. PARALLEL SWEEPS

Consider a $P_x \times P_y \times P_z$ processor layout on a $N_x \times N_y \times N_z$ spatial grid, with integer values for all $\{N_u/P_u\}$ for simplicity. Suppose there are $M$ quadrature directions per octant and $G$ energy groups that can be swept simultaneously. Then each processor must perform $(N_z N_y N_z 8MG)/(P_z P_y P_z)$ cell-direction-group calculations. Aggregate these into tasks, with each task containing $A_x A_y A_z$ cells, $A_m$ directions, and $A_g$ groups. Then each processor must perform $N_{\text{tasks}} \equiv (N_z N_y N_z 8MG)/(A_x A_y A_z A_m A_g P_x P_y P_z)$ tasks. At each stage at least one processor computes a task and communicates to downstream neighbors. The complete sweep requires $N_{\text{stages}} = N_{\text{tasks}} + N_{\text{idle}}$ stages, where $N_{\text{idle}}$ is the number of idle stages for each processor. Parallel sweep efficiency (serial time per unknown / parallel time per unknown per processor) is therefore approximately

$$\epsilon = \frac{T_{\text{task}} N_{\text{tasks}}}{[N_{\text{stages}}]\,[T_{\text{task}} + T_{\text{comm}}]} = \frac{1}{\left[1 + \frac{N_{\text{idle}}}{N_{\text{tasks}}}\right]\left[1 + \frac{T_{\text{comm}}}{T_{\text{task}}}\right]} \ ,$$

where $T_{\text{task}}$ is the time to compute one task and $T_{\text{comm}}$ is the time to communicate after completing a task. In the second line, the term in the first [ ] is $1+$ the pipe-fill penalty and the term in the second [ ] is $1+$ the comm penalty. Aggregating for many small tasks ($N_{\text{tasks}}$ large) minimizes pipe-fill penalty but increases the comm penalty: latency causes $T_{\text{comm}}/T_{\text{task}}$ to increase as tasks become smaller. This assumes the most basic comm model, which can be refined to account for architectural realities (hierarchical networks, random variations, dedicated comm hardware, latency-hiding techniques, etc.).

Traditional KBA chooses $P_z = 1$, $A_m = 1$, $G = A_g = 1$, $A_x = N_x/P_x$, $A_y = N_y/P_y$, and $A_z =$ selectable number of $z$-planes to be aggregated into each task. With $N_k = N_z/(P_z A_z)$, each processor performs $N_{\text{tasks}} = 8MN_k$ tasks. With KBA, $2MN_k$ tasks (two octants) are "launched" from a given corner of the 2D processor layout. For any octant pair the far-corner processor remains idle for the first $P_x + P_y - 2$ stages, so a two-octant sweep completes in $2MN_k + P_x + P_y - 2$ stages. The other two-octant sweeps are similar, so if an octant-pair sweep does not begin until the previous pair's finishes, the full sweep requires $8MN_k + 4(P_x + P_y - 2)$ stages. The KBA parallel efficiency is then

$$\epsilon_{KBA} = \frac{1}{\left[1 + \frac{4(P_x+P_y-2)}{8MN_k}\right]\left[1 + \frac{T_{\text{comm}}}{T_{\text{task}}}\right]} \tag{2}$$

KBA inspires our algorithms, but we do not force $P_z = 1$ or force aggregation factors to have particular values (such as $A_m = 1$), and we allow multiple octants to sweep simultaneously. In contrast to KBA, this requires a scheduling algorithm—a set of rules that tells each processor the order in which to execute tasks when more than one is available. Scheduling algorithms profoundly affect parallel performance, as was noted in [5].

The minimum possible number of stages for given partitioning parameters $\{P_i\}$ and aggregation factors $\{A_j\}$ is $2N_{\text{fill}} + N_{\text{tasks}}$, where $N_{\text{fill}}$ is the minimum number of stages before a sweepfront can reach the center-most processors = number needed to finish a direction's sweep after the center-most processors have finished[5]. If we force $A_x = N_x/P_x$ and $A_y = N_y/P_y$, then

$$N_{\text{fill}} = \frac{P_x + \delta_x}{2} - 1 + \frac{P_y + \delta_y}{2} - 1 + N_k\left(\frac{P_z + \delta_z}{2} - 1\right) \tag{3}$$

$$N_{\text{idle}}^{\text{min}} = P_x + \delta_x - 2 + P_y + \delta_y - 2 + N_k(P_z + \delta_z - 2) \tag{4}$$

where $\delta_u = 0$ or 1 for $P_u$ even or odd, respectively, and

$$N_{\text{stages}}^{\text{min}} = N_{\text{idle}}^{\text{min}} + (8MGN_k)/(A_m A_g) \ , \tag{5}$$

Important observation: $N_{\text{idle}}^{\text{min}}$ is lower for $P_z = 2$ than for the KBA choice of $P_z = 1$, for a given $P$. In both cases $P_z + \delta_z - 2 = 0$, but with $P_z = 2$, $P_x + P_y$ is lower. As we explain in Section 3.3, the benefit of $P_z = 2$ lets you double $N_z$ and $P_z$ with no increase in stage count.

Given the minimum stage count for a sweep, we can see that the optimal efficiency is:

$$\epsilon_{opt} = \frac{1}{\left[1 + \frac{P_x+\delta_x+P_y-4+\delta_y+N_k(P_z+\delta_z-2)}{8MGN_k/(A_m A_g)}\right]\left[1 + \frac{T_{\text{comm}}}{T_{\text{task}}}\right]} . \tag{6}$$

It is not obvious that any schedule can achieve the lower bound of Eq. (5), because "collisions" of the $8M$ sweepfronts force processors to delay some fronts by working on others. Bailey and Falgout described a "data-driven" schedule that achieved this minimum stage count in some tests, but they left open the question of the conditions under which this schedule would be guaranteed to achieve the minimum[5].

### 2.1. Observations About Aggregation

*Here, we will explain some things about aggregation, why we choose $A_u$ as we do, why we heavily use $P_z = 2$ but don't mention $P_y = 2$, etc.*

## 3. PROOFS OF OPTIMAL SCHEDULING

In this section we describe a family of scheduling algorithms that we have found to be "optimal" in the sense that they complete the full eight-octant sweep in the minimum possible number of stages for a given $P_u$ and $A_j$. For one such algorithm—the "depth-of-graph" algorithm, which gives priority to the task that has the longest chain of dependencies awaiting its execution—we sketch our proof of optimality. For another—the "push-to-central" algorithm, which prioritizes tasks that advance a wavefront to a specified central plane in the processor layout—we simply describe the scheduling rules but do not prove optimality. We then observe that these two algorithms are the two endpoints of one-parameter family of algorithms, each of which should execute sweeps with the minimum stage count. In a later section our computational results show optimality for both the depth-of-graph and push-to-central algorithms, but we do not show results for other members of the optimal family.

To facilitate the discussion and proofs that follow, let us define

$$i \in (1, P_x) = \text{the } x \text{ index into the processor array}, \tag{7}$$

with similar definitions for the $y$ and $z$ indices, $j$ and $k$. We will also use

$$X = \frac{P_x + \delta_x}{2} \; , \quad Y = \frac{P_y + \delta_y}{2} \; , \quad Z = \frac{P_z - \delta_z}{2} \tag{8}$$

to define "sectors" of the processor array, e.g. ($i \in (1, X),\ j \in (1, Y)$),
($i \in (X+1, P_x),\ j \in (1, Y),\ k \in (Z+1, P_z)$), etc. These sectors relate to our scheduling algorithm.

We will use superscripts to represent octants/quadrants, e.g. $^{++}$ to denote ($\Omega_x > 0,\ \Omega_y > 0$), $^{-+-}$ to denote ($\Omega_x < 0,\ \Omega_y > 0,\ \Omega_z < 0$), etc.

The details of the depth-of-graph algorithm will become clear in the proofs that follow. The push-to-central algorithm prioritizes tasks according to the following rules.

1. If $i < X$, then tasks with $\Omega_x > 0$ have priority over tasks with $\Omega_x < 0$, while for $i > X$ tasks with $\Omega_x < 0$ have priority.
2. If multiple ready tasks have the same $\Omega_x$, then for $j < Y$ tasks with with $\Omega_y > 0$ have priority, while for $j > Y$ tasks with $\Omega_y < 0$ have priority.
3. If multiple ready tasks have the same $\Omega_x$ and $\Omega_y$, the for $k < Z$ tasks with $\Omega_z > 0$ have priority, while for $k > Z$ tasks with $\Omega_z < 0$ have priority.

Note that this schedule pushes tasks toward the $i = X$ central processor plane with top priority, followed by pushing toward the $j = Y$ (second priority) and $k = Z$ (third priority) central planes.

The depth-of-graph and push-to-central algorithms differ only in regions of the processor-layout domain in which the "depth" priority differs from the "central" priority for some octants. In those regions for those octants, one can view the two algorithms as differing only in the degree to which they allow the two opposing octants' tasks to interleave with each other. The push-to-central algorithm maximizes this interleaving while the depth-of-graph algorithm minimizes it. One can vary the degree of interleaving between these extremes to create other scheduling algorithms. Our preliminary analysis (not shown here) indicates that all of these algorithms achieve the minimum possible stage count.

In the following subsections we describe our proof of optimality for the depth-of-graph algorithm.

## 3.1. Depth-of-Graph Algorithm: General

The essence of the depth-of-graph scheduling algorithm is that each processor gives priority to tasks with the most downstream dependencies, or the greatest remaining *depth of graph*. (By "graph" we mean the task dependency graph, as pictured in Fig. 1.) This quantity, which we will denote $D(O)$ for an octant $O$, is a simple function of processor location and octant direction. The depth-of-graph algorithm prioritizes tasks according to the following rules.

1. Tasks with higher $D$ have higher priority.
2. If multiple ready tasks have the same $D$, then tasks with $\Omega_x > 0$ have priority.
3. If multiple ready tasks have the same $D$ and $\Omega_x$, then tasks with $\Omega_y > 0$ have priority.
4. If multiple ready tasks have the same $D$, $\Omega_x$, and $\Omega_y$, then tasks with $\Omega_z > 0$ have priority.

We will develop our proof with the aid of indexing algebra, but the core concept stems from Eq. (6). The formula for $\epsilon_{opt}$ implies that three conditions are sufficient for a schedule to be optimal:

1. The central processors must begin working at the earliest possible stage.
2. The highest priority task must be available to the central processors at every stage (i.e., once a central processor begins working, it is not idle until all of its tasks are completed).
3. The final tasks completed by the central processors must propagate freely to the edge of the problem domain.

If these three criteria are met, a schedule will be optimal as defined by Eq. (6). For $P_z = 1$ and $P_y > 1$ the four central processors are defined by $i = X$ or $X + 1$ and $j = Y$ or $Y + 1$. For $P_z > 1$ the eight central processors are defined by $i = X$ or $X + 1$, $j = Y$ or $Y + 1$, and $k = Z$ or $Z + 1$.

The “corner” processors begin working at the first stage. This leads to the first condition being satisfied, for the four or eight sweep fronts (for $P_z = 1$ or $> 1$) proceed unimpeded to the four or eight central processors, with no scheduling decisions required. The second condition is not obvious, but we will demonstrate that the depth-of-graph prioritization causes it to be met. It is likely that any scheduling algorithm that satisfies item (2) will achieve item (3). We will show that depth-of-graph does.

We will examine the behavior of the depth-of-graph scheduling algorithm within three separate partitioning and aggregation schemes. The first, $P_z = 1$, uses the same partitioning as KBA; however, as mentioned, we do not impose the same restrictions on our aggregation, and we launch tasks for all quadrants simultaneously. The second arrangement is a doubling of this, with $P_z = 2$. We refer to the $P_z = 2$ configuration as a “hybrid” decomposition, since it shares traits with both the $P_z = 1$ case and the $P_z > 2$ case. We call the latter “volumetric”, since it decomposes the domain into regular, contiguous volumes.

Since the basic scheme of our algorithm sets priorities based on *downstream* depth of graph, we will use $D(O)$ to represent this quantity for octant $O$:

$$\begin{array}{rlll} D(+-) = & (P_x - i) & +(j-1) & \\ D(-+-) = & (i-1) & +(P_y - j) & +(k-1) \end{array} . \tag{9}$$

Since much of the algebra for stage counts depends on the depth of a task *into* the task graph, let us define a direction-dependent variable $s$:

$$\begin{array}{rlll} s^{++} = & (i-1) & +(j-1) & \\ s^{--+} = & (P_x - i) & +(P_y - j) & +(k-1) \end{array} \tag{10}$$

etc.

Notice that $D$ and $s$ are related by $D(O) + s(O) =$ total depth of graph $= P_x + P_y + P_z - 3$. We find that $D$ is convenient for discussing priorities, and $s$ is a convenient measure for the stage a task will be executed.

Our aggregation factors determine what we cluster together as a single task. The spatial aggregation factors, $A_x$, $A_y$, and $A_z$, determine how many cells make up a cellset. The factors $A_m$ and $A_g$ group angles into anglesets and groups into groupsets. A task is then the computation for a single cellset, for a single energy groupset, for a single angleset.

We will use $N_t$ to represent the number of tasks per processor per quadrant for $P_z = 1$ and per octant for $P_z > 1$. This is a different quantity from $N_{\text{task}}$ discussed above; it takes one fourth the value if $P_z = 1$ and one eighth otherwise.

Let

$$m \in (1, N_t) = \text{specific task from the ordered list for a quadrant,} \tag{11}$$

and $\mu$ represent the stage at which a task is completed. Thus,

$$\mu_{pq}(m^{rs}, i, j) = \text{stage at which processor } (i,j) \text{ in sector } pq \text{ performs task } m \text{ in quadrant } rs \tag{12}$$

### 3.2. $P_z = 1$ Decomposition

For this type of decomposition we aggregate using $A_x = N_x/P_x$ and $A_y = N_y/P_y$. In the next section, we will discuss how $A_z$ and $A_m$ are optimized based on a performance model, but for now they are treated as free variables. We have defined $N_t = (N_z/A_z)(2M/A_m)(G/A_g)$, which encapsulates the multiple cellsets, group-sets, and direction-sets for a processor. Since the values for $A_x$ and $A_y$ ensure that every completed task will satisfy dependencies downstream, our analysis will not directly involve aggregation factors.

The foundation of the scheduling algorithm we are analyzing is downstream depth of graph, so let us take a look at this. $D(O)$ depends on processor location, and different regions of the problem domain will have different priority orderings. These regions can be determined by index algebra.

### 3.2.1. Priority regions

Since we assign quadrants priorities based on $D(O)$, which is a simple function of $i$ and $j$, it is a simple matter to determine beforehand what a processor's priorities will be. We find that the problem domain is divided into distinct, contiguous regions with definite priorities. For example, processor $(1, 1)$ executes tasks in the order $++$, $+-$, $-+$, $--$. At times we will find it convenient to refer to a region by its priority ordering. It is also convenient to refer to a quadrant as a region's primary, secondary, etc.

The boundaries between regions of different priorities are planes (or lines in our 2D processor layout) defined by the solutions to the equations $D(O_1) = D(O_2)$ for distinct octants (or quadrants) $O_1$ and $O_2$. For example, let us examine quadrants $++$ and $+-$.

$$\begin{aligned} D(++) &= D(+-) \\ (P_x - i) + (P_y - j) &= (P_x - i) + (j - 1) \\ P_y &= 2j - 1 \\ (P_y + \delta_y) - \delta_y + 1 &= 2j \\ j &= Y + \frac{1 - \delta_y}{2} \end{aligned} \tag{13}$$

If $P_y$ is even, this is $j = Y + \frac{1}{2}$; if $P_y$ is odd, it is $j = Y$. The non-integer value means that the plane passes *between* processors, while an integer value means that the plane passes *through* processors. Thus, the even case cleanly divides the problem domain into two regions: $j < Y$, where $++$ quadrants have priority, and $j > Y$, where $+-$ quadrants have priority. The $P_y$ odd case leaves us with a tie for processors at $j = Y$. We use a simple scheme here: the first tie-breaker goes to tasks with $\Omega_x > 0$, and the second tie-breaker is for $\Omega_y > 0$. This is why we've chosen to round the values of $X$ and $Y$ up rather than down.

Note that Eq. (13) is also the solution of $D(-+) = D(--)$. There is an analagous plane bounding the two quadrant pairs with differing signs in $\Omega_x$, given by

$$i = X + \frac{1 - \delta_x}{2}. \tag{14}$$

There are also two quadrant pairs with sign differences in both $\Omega_x$ and $\Omega_y$. Solving for these boundaries, we find

$$\begin{aligned} D(++) &= D(--) \\ (i-X)+(j-Y) &= \frac{1-\delta_x}{2} + \frac{1-\delta_y}{2} \end{aligned} \tag{15}$$

and

$$\begin{aligned} D(+-) &= D(-+) \\ (i-X)-(j-Y) &= \frac{1-\delta_x}{2} - \frac{1-\delta_y}{2}. \end{aligned} \tag{16}$$

We can see that Eq. (15) is a line of slope $-1$ through the center of the domain, with a possible shift of one half or one. If $P_u$ are both even, then we have processors with a tie. Again, we apply our tie-breaker, and these processors can be thought of as belonging to the region that prioritizes the winning quadrant. If one of $P_u$ is odd, then there are no ties, and what we have is a stair-stepped boundary. If both of $P_u$ are odd, we again have a straight line with ties. The situation is similar for Eq. (16). These regions are illustrated in Fig. 3.2.1 for all combinations of odd and even $P_u$.

It is worth noting that the processor layouts in the figure could be either the entire domain or a small central subset; the lines and all they signify are the same either way.

We call attention to the "central" processors, which are shaded in the figure. Any given central processor has one clear primary quadrant that takes priority over others. For a given central processor, call this quadrant "$P$." The two central processors immediately adjacent to the given processor are downstream for quadrant $P$, and this quadrant is either the second or third priority for each of these. The fourth central processor has quadrant $P$ as its lowest priority.

### 3.2.2. Filling the pipe

At the outset of a sweep, there are only four processors that have all of their incoming fluxes. These are the corner processors, for which boundary conditions satisfy the dependencies. Each of these processors completes its first task at stage one, which satisfies the dependencies for its downstream neighbors, and thus begin the waves of task-flow that we call the sweep.

Let us examine the order in which tasks in quadrant $++$ are completed in sector $--$. We begin at stage $\mu = 1$ with processor $(i, j) = (1, 1)$ performing task $m = 1$. Once this is completed (i.e. in stage 2), processors $(1, 2)$ and $(2, 1)$ can perform task 1, and processor $(1, 1)$ moves on to task 2. In stage 3, processors $(1, 3)$, $(2, 2)$, and $(3, 1)$ perform task 1, processors $(1, 2)$ and $(2, 1)$ perform

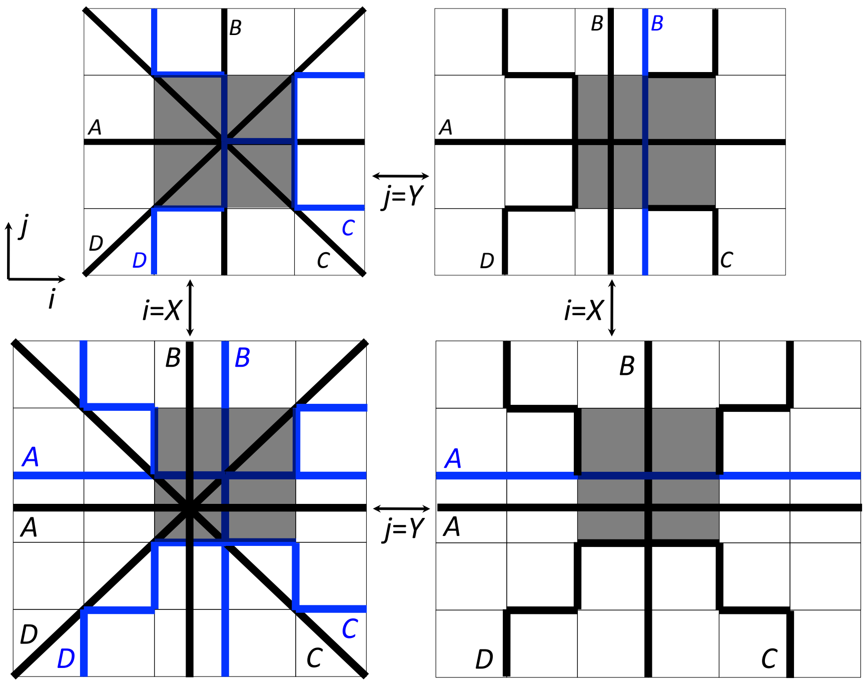

**Figure 2.** Priority regions for $P_x$-$P_y$ even-even, odd-even, even-odd, and odd-odd. Lines $A$-$D$ represent Eqs. (13-16), respectively, with blue versions after tie-breaker. Central processors are shaded. Figures are "zoomed in" on the central portion of the processor domain, which may have arbitrarily large extent.

task 2, and processor $(1,1)$ performs task 3. We can generalize this pattern with the simple expression

$$\mu(m^{++}, i, j) = (i-1) + (j-1) + m^{++} = s^{++} + m^{++}. \tag{17}$$

This equation describes a wave emanating from the corner, but rather than spherical this wave-front is planar. Viewed in our 2D processor layout, it appears as a line perpendicular to the tasks' direction.

The procession of tasks proceeds in exactly this way from each corner, with each sector performing tasks from its primary quadrant, until there is a processor with multiple tasks available. Thus, we find that

$$\mu(m^{+-}, i, j) = (i-1) + (P_y - j) + m^{+-} = s^{+-} + m^{+-}, \tag{18}$$

$$\mu(m^{-+}, i, j) = (P_x - i) + (j-1) + m^{-+} = s^{-+} + m^{-+}, \tag{19}$$

and

$$\mu(m^{--}, i, j) = (P_x - i) + (P_y - j) + m^{--} = s^{--} + m^{--}. \tag{20}$$

### 3.2.3. Starting on the central processors

As can be seen from Eqs. (13-16) or Fig. 3.2.1, each quadrant has top priority for one entire sector, so even if other tasks are available, these stage counts will hold within the initial sector. Thus, the central processors are reached in $X + Y - 1$ stages, just as in Eq. (3), which satisfies the first condition for optimality.

This result also gives us a start on condition (2). It is clear from Eqs. (17-20) that successive tasks in a given quadrant take place at successive stage counts. If the central processor gets its first task at stage $\mu$, it will receive the second at $\mu + 1$, etc. This guarantees that all of the tasks in a central processor's highest priority quadrant will arrive in sequence, allowing the processor to stay busy as it processes its first quadrant.

This also satisfies in advance half of the dependencies for each central processor's second- and third-priority quadrants. As an example, take processor $(X, Y)$. As it computes its $++$ tasks, $(X, Y+1)$ and $(X+1, Y)$ are computing their $+-$ and $-+$ tasks and communicating to $(X, Y)$. This satisfies half of the dependencies for these two quadrants; for example, tasks in the $+-$ octant cannot be executed at $(X, Y)$ until $(X-1, Y)$ has computed them too. This is addressed next.

### 3.2.4. Dependencies for second- and third-priority quadrants

Let us continue with our example, focusing on the $++$ tasks and their progress. We know that $(X, Y)$ is delivering tasks to $(X, Y+1)$, but the latter still needs the solution from $(X-1, Y+1)$. This second chain of dependencies fundamentally hinges on $(1, Y+1)$, since this is the farthest upstream task that we haven't already established. Similarly, $(X+1, 1)$ is the key for $(X+1, Y)$. Let us examine these processors.

The first $++$ task is ready for processor $(1, Y+1)$ at stage

$$\mu = s^{++} + m = (1-1) + ([Y+1]-1) + 1 = Y + 1\ . \tag{21}$$

However, that processor is dedicated to its $+-$ tasks until they are exhausted, so the $++$ tasks experience a delay. The value of this delay is

$$\begin{aligned} d &= \mu(N_t^{+-}+1,1,Y+1) - \mu(1^{++},1,Y+1) \\ &= [(1-1)+(P_y-[Y+1])+(N_t+1)] - [(1-1)+([Y+1]-1)+1] \\ &= (P_y-2Y)+N_t-1 = N_t-1-\delta_y \ . \end{aligned} \tag{22}$$

Similarly, we find that the delay at $(X+1,1)$ is $d = N_t - 1 - \delta_x$. (Note: It is possible for this delay to equal zero, if $N_t = 2$ and $\delta_u = 1$. The equations still hold in this case; there is simply no actual delay.)

Turning back to Eqs. (17-20), we see that the final $+-$ task progresses smoothly from $(1, Y+1)$ to $(X, Y+1)$, leaving behind it no competition for our $++$ tasks. Since the dependencies have already been satisfied along $j = Y$, the first $++$ task at $(1, Y+1)$ initiates another smooth wave. That is to say, the same delay applies to all of the $++$ tasks in the region with priorities $(+-, ++, ...)$, so the second dependency of $(X, Y+1)$ will be satisfied at stage

$$\begin{aligned} \mu = s^{++} + m + d &= [(X-1)+([Y+1]-1)] + [1] + [N_t-1-\delta_y] \\ &= X+Y+N_t-1-\delta_y \ . \end{aligned} \tag{23}$$

Processor $(X, Y+1)$ puts quadrant $++$ either second or third. If second, then the schedule demands that the first $++$ task be executed at stage

$$\begin{aligned} \mu = s^{+-} + (N_t+1) &= (X-1)+(P_y-[Y+1])+N_t+1 \\ &= X+Y+N_t-1-\delta_y \ , \end{aligned} \tag{24}$$

and we see that the timing is perfect, noting that each stage will bring the next task. If $++$ tasks are third priority, then the processor won't want them for an additional $N_t$ stages, and the dependencies are clearly satisfied.

The logic and algebra are identical for processor $(X+1, Y)$. If it prioritizes $++$ tasks second, it expects them at stage $\mu = X + Y + N_t - 1 - \delta_x$. Regardless of its priorities, the $++$ tasks arrive starting at $\mu = X + Y + N_t - 1 - \delta_x$, and the processor is always able to execute its highest priority task.

The example quadrant we just examined wins the tie-breaker. To see how the losing quadrants progress, let us look at $+-$ tasks in processor $(X, Y)$. If $+-$ tasks are its second priority, this processor will expect them at stage

$$\mu = s^{++} + N_t + 1 = X + Y + N_t - 1 \ .$$

This task depends on work from processors $(X, Y+1)$ and $(X-1, Y)$, which must be complete by stage $X + Y + N_t - 1$. We find that the dependency from $(X, Y+1)$ is met at stage

$$\mu = s^{+-} + 1 = (X-1) + (P_y - Y) + 1 = X + Y - \delta_y \ ,$$

which is necessarily less than $X + Y + N_t - 1$.

Task execution at $(X-1, Y)$ rests on the upstream processor $(1, Y)$. Computing, as we did before, the delay that $+-$ tasks experience due to $++$, we find $d = N_t - 1 + \delta_y$. Thus, the dependency will be satisfied at stage

$$\begin{aligned}\mu = s^{+-} + m + d &= [(X-1) + (P_y - Y)] + 1 + [N_t - 1 + \delta_y] \\ &= X + Y + N_t - 1 \ ,\end{aligned}$$

which again is perfect timing. Given the flow of tasks to the primary central processor, into the secondary regions, and to the secondary central processors, every central processor has the dependencies for its secondary and tertiary tasks satisfied in time for execution with no idle time.

### 3.2.5. Emptying the pipe

The lowest-priority quadrant of a central processor depends on the two central processors just upstream. We have established that the central processors execute with no idle time through their third-priority quadrants, which means that they have finished their neighbor's fourth-priority quadrants. Thus, the dependencies for each central processor's current highest priority task are shown to be satisfied at every stage of the sweep. The depth-of-graph algorithm therefore satisfies the second condition of an optimal schedule.

To assist in understanding the dynamics of task flow, we sketch the 20 stages of the sweep evolution for the case of $N_t = 3$, $P_x$ odd, and $P_y$ odd. Figure (3.2.5) shows this for the central $5 \times 5$ set of processors.

In order for the third condition to be met, then for the final quadrant that is executed by a central processor, as the wave of tasks proceeds to the final corner (a) there must be no competition with higher priority tasks, and (b) the upstream dependencies must be satisfied. We have shown how tasks proceed; each is by now waiting at the boundaries to its final sector, and though they might be only one step ahead, no other task will impede the lowest priority quadrant in a sector.

As can be seen in the progression of a sweep (Fig. 3.2.5), the limiting factor for the final quadrant to enter a sector is that sector's central processor itself. As the furthest upstream task for the quadrant in its priority-region, it is a prerequisite for the rest of the sector to execute those tasks. To execute to completion without delays only requires that we have the standard planar propagation; usually, the downstream dependencies are ahead of schedule.

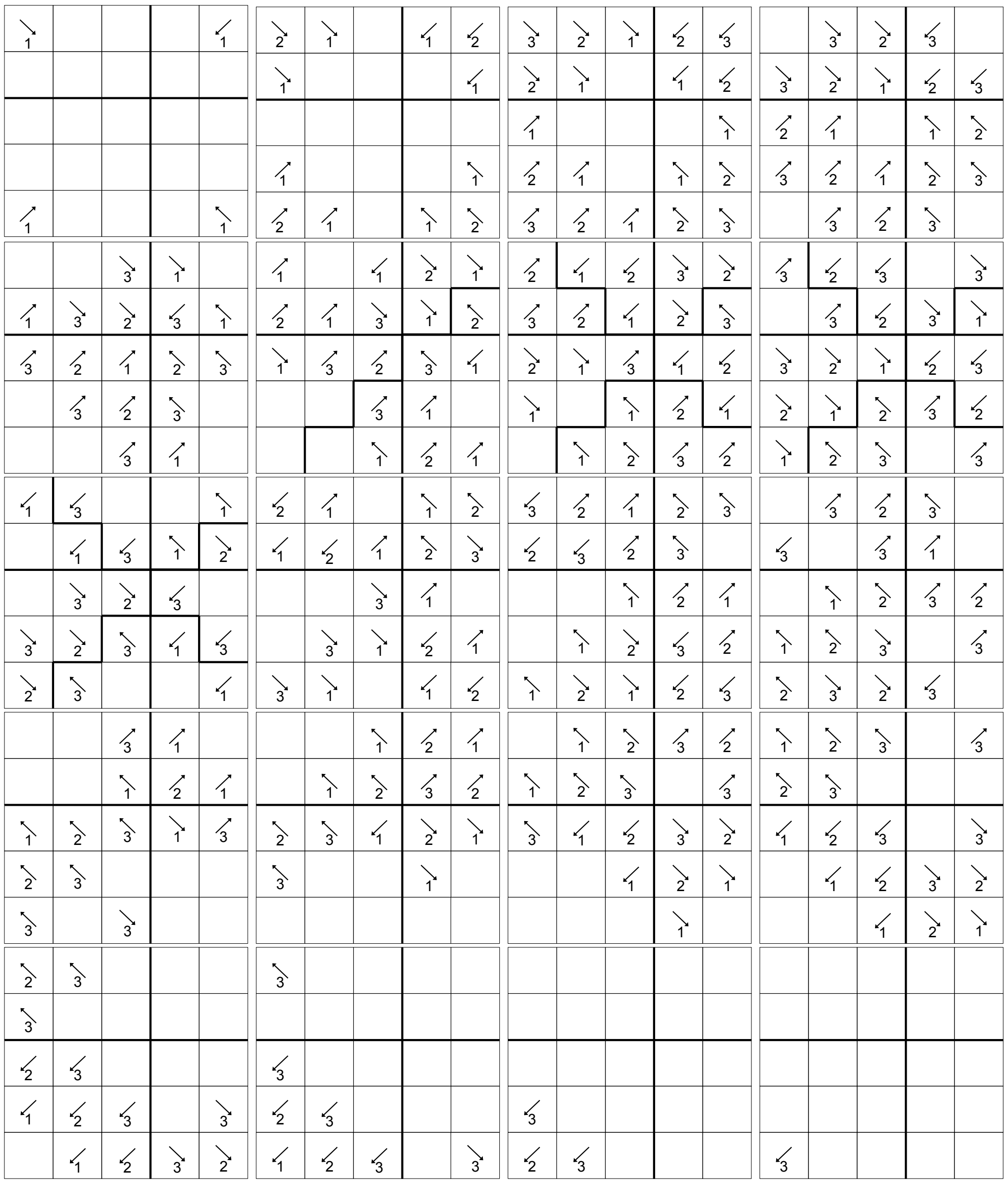

**Figure 3.** Illustration of 20 consecutive stages of a sweep scheduled by the depth-of-graph scheduler, for the case of $P_x$ and $P_y$ both odd, using $N_t = 3$ for illustration. Numbers represent tasks for a given quadrant.

## 3.3. $P_z = 2$ Decomposition (Hybrid)

Now consider the case of $P_z = 2$. Above, we considered task groups in terms of quadrants, which are more properly sets of two octants. We did not specify the ordering of tasks within a quadrant because the proof holds true regardless of that order. Thus, we are free to do the entire "upward" ($\Omega_z > 0$) octant first, followed by the entire downward octant, and all of the properties we have established above are unchanged.

In the hybrid decomposition, we schedule tasks for the $k = 1$ processors exactly this way, and we schedule the $k = 2$ processors in reverse. While the lower processors solve their upward tasks, the upper processors solve their downward tasks, so that by the time one is done, the other is waiting. All other scheduling concerns are handled exactly the same as in $P_z = 1$.

This leads to a surprising result: if we were to take a $P_z = 1$ problem and then double both $N_z$ and $P_z$, then the task flow for the $k = 1$ processors would be indistinguishable from the $P_z = 1$ case. They work their $\Omega_z > 0$ octants, as before, and then their $\Omega_z < 0$ octants, just as before. This means that using the hybrid decomposition instead of the $P_z = 1$ allows for doubling the number of cells in $z$ and doubling the number of processors *with no increase in solve time*.

This benefit rests on the initial step of computation. In $P_z = 1$, only four processors had tasks with no unsatisfied dependencies; now all eight corner processors launch their primary octants at once. We experience the same pipe-fill penalty as before, and the number of tasks per processor is the same. We simply perform twice the work in the same time.

### 3.4. $P_z > 2$ Decomposition (Volumetric)

In this section, we will examine what we call a volumetric decomposition, which is the full extension of the $P_z = 1$ decomposition into three dimensions. The same requirements for optimality apply, and the task flow follows the same principles.

#### 3.4.1. Pipe-fill

Things begin much as before, with eight corner processes initiating waves of tasks in their primary octants. Stage counts are the same,

$$\mu(m, i, j, k) = s + m \ , \tag{25}$$

and they carry the same implications about the central processors.

The task waves still meet at the priority-region boundaries and complete all of their primary tasks before turning to secondary work. The tail end of the primary task waves still open up unchallenged space, and the now secondary task waves cross over and propagate in a smooth planar wave. The delay term is the same: $d = N_t - 1 \pm \delta_u$, with plus/minus for losing/winning the tie-breaker. The stage count to reach the central processor across the border is still exactly what is needed. For example, arrival of $---$ tasks at processor $(X+1, Y, Z+1)$ is given by

$$\begin{aligned}\mu &= s^{---} + m + d \\ &= [(P_x - [X+1] + (P_y - Y) + (P_z - [Z+1])] + 1 + [N_t - 1 + \delta_y] \\ &= (P_x - X) + (P_y + \delta_y - Y) + (P_z - Z) + N_t - 2 \\ &= (P_x - X) + Y + (P_z - Z) + N_t - 2\ ,\end{aligned}$$

while the stage this processor begins its secondary tasks is

$$\begin{aligned}\mu &= s^{-+-} + m = (P_x - [X+1]) + (Y-1) + (P_z - [Z+1]) + (N_t + 1) \\ &= (P_x - X) + Y + (P_z - Z) + N_t - 2\ .\end{aligned}$$

Again, it's as smooth as clockwork.

### 3.4.2. Priority regions

Much as before, the domain is divided into regions with different priority orders based on relative depths of graph for different octants. For $P_z = 1$, there were six distinct pairs of quadrants and eight regions of different priority orders. For $P_z > 2$, there are 28 distinct octant pairs $(7 + 6 + ... + 1)$, and, as we will see, 96 different regions. The regions are defined by the planes at which two octants have equal depth of graph.

For $P_z = 1$ or $P_z = 2$, after the primary quadrant there were two quadrants advancing in each sector. For $P_z > 2$ there are three octants entering each sector, which we will nickname $R$, $B$ and $G$ (for red, blue and green). We will call the primary octant $P$. The priority regions $(P, R, ...)$, $(P, B, ...)$ and $(P, G, ...)$ are defined by planes that we will call the $RB$, $RG$ and $BG$ boundaries, given by $D(R) = D(B)$, etc. and illustrated in Fig.3.4.2. Because these three planes intersect along a single line (perpendicular to the figure), they divide the sector into six regions. Each octant has the highest priority in two of these regions.

Whereas for $P_z = 1$ the secondary and tertiary quadrants finished a sector before the final quadrant moved in, for $P_z > 2$ we see six octants at play in a sector. The three octants with directions opposite $R$, $B$ and $G$, which we will call $\bar{R}$, $\bar{B}$ and $\bar{G}$, arrive before the first three are finished. The boundary planes for each of these with the $RBG$ octants that are not its inverse are the sector boundaries. The boundary planes with their opposites are perpendicular to the problem domain's diagonals ; each of these carves up two of the six regions where the second octant of $RBG$ was unchallenged.

Throughout this choreography, the fundamental features of the scheduling algorithm hold. The central processors get their work as early as possible, they stay busy until they are done, and their final tasks fly free to the end of the problem.

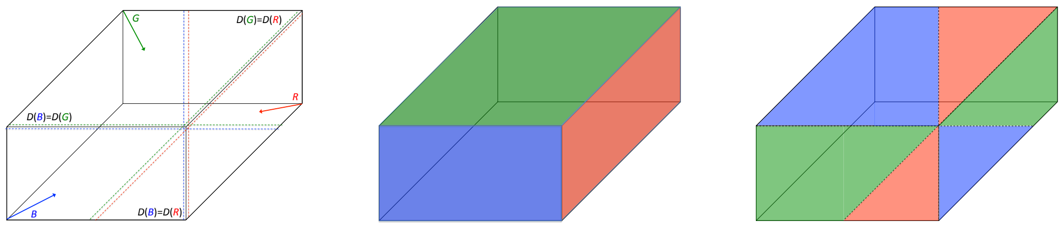

**Figure 4.** (a) Planes of equal depth of graph divide a sector into regions of different priority. (b) Each octant has second priority in two of the six regions. (c) Each octant has third priority in two disjointed regions.

## 4. OPTIMAL SWEEPS

Here we describe how we have used our optimal *scheduling* algorithm to generate an optimal *sweep* algorithm. Given an optimal schedule we know exactly how many stages a complete sweep will take, and thus can estimate the parallel efficiency of a sweep with such a schedule:

$$\epsilon_{opt} = \frac{1}{\left[1 + \frac{P_x+\delta_x+P_y-4+\delta_y+N_k(P_z+\delta_z-2)}{8MGN_k/(A_m A_g)}\right]\left[1+\frac{T_{\text{comm}}}{T_{\text{task}}}\right]}. \tag{26}$$

Given Eq. (26) we can choose the $\{P_i\}$ and $\{A_j\}$ that maximize efficiency and thus minimize total sweep time. This optimization over $\{P_i\}$ and $\{A_j\}$, coupled with the scheduling algorithm that executes the sweep in $N_{stages}^{min}$ stages, yields what we call an *optimal sweep* algorithm.

It is interesting to compare $\epsilon_{\text{KBA}}$ to $\epsilon_{opt}$, especially in the limit of large $P$ (which allows us to ignore the $\delta_u$ and the numbers 2 and 4 that appear in the equations). In the large-$P$ limit, with $P_x + P_y \approx P^{1/2} + P^{1/2}$, Eq. (2) becomes

$$\epsilon_{\text{KBA}} \to \frac{1}{\left[1+\frac{4(2P^{1/2})}{8MN_z/A_z}\right]\left[1+\frac{T_{comm}}{T_{task}}\right]} = \frac{1}{\left[1+\frac{P^{1/2}}{MN_z/A_z^{\text{KBA}}}\right]\left[1+\frac{T_{comm}}{T_{task}}\right]} \tag{27}$$

Now consider $\epsilon_{opt}$ with $G = 1$, $P_z = 2$, and $P_x + P_y \approx 2(P/2)^{1/2}$. For comparison we aggregate to the same number of tasks as in KBA (which is likely sub-optimal) by setting $A_m = 1$ and $A_z = A_z^{\text{KBA}}/2$. The result is

$$\epsilon_{opt} \to \frac{1}{\left[1+\frac{\sqrt{2}P^{1/2}}{8MN_z/A_z^{\text{KBA}}}\right]\left[1+\frac{T_{comm}}{T_{task}}\right]}. \tag{28}$$

An interesting question is how many more processors the optimal schedule can use with the same efficiency as KBA. We see that with the chosen $\{P_i\}$ and (sub-optimal) $\{A_j\}$, the optimal algorithm produces the same efficiency as KBA with $(8/\sqrt{2})^2 = 32$ times as many processors. That is, even without optimizing the $\{A_j\}$, this scheduling algorithm yields the same efficiency on 132k cores as traditional KBA on 4k cores. Optimizing the $\{A_j\}$ can improve this even further.

Our performance model is Eq. (26) with the following definitions:

$$T_{comm} = M_L T_{latency} + T_{byte} N_{bytes} \tag{29}$$

$$T_{task} = A_x A_y A_z A_m A_g T_{grind} \tag{30}$$

where $T_{latency}$ is the message latency time, $T_{byte}$ is the additional time to send one byte of message, $N_{bytes}$ is the total number of bytes of information that a processor must communicate to its downstream neighbors at each stage, and $T_{grind}$ is the time it takes to compute a single cell, direction, and energy group. $N_{bytes}$ is calculated based on the aggregation and spatial discretization scheme; the other parameters are obtained through testing. We use the parameter $M_L$ to explore performance as a function of increased or decreased latency. If we find that a high value of $M_L$ is needed for our model to match our computational results, then we look for things to improve in our code implementation.

We have implemented in our PDT code an “auto” partitioning and aggregation option. When this option is engaged, the code uses empirically determined numbers for $T_{latency}$, $T_{byte}$, $N_{bytes}$, and $T_{grind}$ for the given machine, then for the given problem size it searches for the combination of $P_u$ and $A_j$ that minimize $\epsilon_{opt}$. This relieves users of the burden of choosing all of these parameters. In the numerical results shown in the following section we did not employ this option, because we were exploring variations in performance as a function of aggregation parameters and thus wanted to control them. However, we often use this option when we use the PDT code to solve practical problems.

We remark that it takes much more than a good parallel algorithm to achieve the excellent scaling results that we provide in the following section. Implementation details are extremely important for any code that attempts to scale to $10^5$ processors and beyond. Our results are due in no small part to the STAPL library, on which the PDT code is built. STAPL provides parallel data containers, handles all communication, and much more. See [8–10] for more details.

## 5. COMPUTATIONAL RESULTS

Here, we present the results of a series of scaling tests. The problem being solved is based on the scaling series used by Zerr, described in depth in Section 5.3 of [3]. The problem is a cube composed of five different materials, each with a different source strength and cross section. As $P$

grows, the problem domain is stretched, but cell size, optical thickness, and $N_u/P_u$ are unchanged. The problem has only one energy group, 80 quadrature directions, and 4096 cells per core. A problem with more groups and angles would exhibit higher parallel efficiencies.

We used a weak scaling scheme, holding constant the number of unknowns per processing unit. We ran this series from $P = 1$ to $P = 131,072$, with the scheduling algorithm analyzed above, with a related but alternate optimal schedule, and with a non-optimal version that simply executes tasks in the order received. The problems were run on the Sequoia machine at LLNL, which is a BG/Q architecture with 16 cores per node.

All results are for $P_z = 2$, and efficiencies are based on solve times normalized to a $P = 8$ run. The solve times do not include setup, but they *do* include communications and all of the computation for converging the iterations to a tolerance of $0.1\%$.

A can be seen in Fig. 5, the optimal schedules perform very efficiently out to large core counts. We have also compared the observed stage counts and against the minimum-possible stage counts described previously in this paper, and in *every* case they agree exactly, for both of the optimal scheduling algorithms. As the figures show, the non-optimal schedule does not perform as well as the optimal schedules, but it degrades surprisingly slowly.

Figure 5 also shows the efficiency predictions of our performance model, for two different latency multipliers. The "low-latency" plot used $M_L = 4$ in Eq. (29), which is what we would hope to achieve in a nearly perfect implementation of our algorithms in our code. The "high-latency" plot used $M_L = 70$, and it agrees quite closely with our observed performance. This suggests that there may be code implementation details that we can improve.

Figure 5 provides results out to 393k cores for a newer version of our code. This version is not yet as efficient as the older version, but it is better designed and should outperform the older version once a few details are addressed. Even so, it achieves approximately 60% parallel efficiency at 393k cores. This is a remarkable achievement for transport sweeps in a one-group problem.

## 6. CONCLUSIONS

Sweeps can be executed efficiently at high core counts. One key is an optimal scheduling algorithm that executes simultaneous multi-octant sweeps with the minimum possible idle time. Another is partitioning and aggregation factors that minimize total sweep time. An ingredient that helps to attain this is a performance model the predicts performance with reasonable quantitative accuracy. Of course, none of this is sufficient to attain excellent parallel efficiency without great care in implementation. But with all of these ingredients in place, sweeps can be executed with high efficiency to core counts of $10^5 - 10^6$.

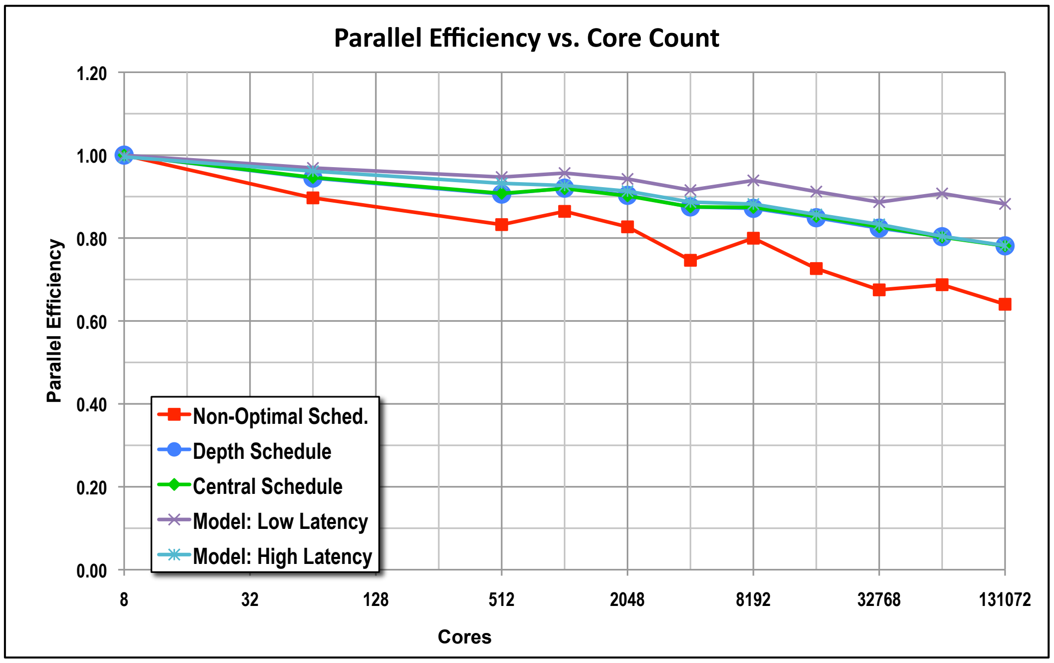

**Figure 5.** Weak scaling performance with PTTL.

Our results demonstrate this clearly. They also show that at least two different sweep scheduling algorithms achieve the minimum possible stage count, in exact agreement with our theory and "proof." The common perception that sweeps do not scale beyond a few thousand cores is simply not correct. Even with a relatively small problem (1 energy group, 80 total directions, and 4096 cells per core) our PDT/STAPL code has achieved approximately 60% efficiency on 393,216 cores. With more energy groups and directions this efficiency would increase. Further, we have reason to believe that we can improve this even further by addressing some implementation details.

The analysis and results in this summary are for 3D Cartesian grids with "brick" cells. We are working on sweeps for AMR-brick grids, for nuclear-reactor grids that resolve pin geometries, and for arbitrary polyhedral-cell grids. For some grids there may be efficiency gains if processors are allowed to "own" non-contiguous cell-sets, an option considered in [5] with the label "domain overloading." Reflecting boundaries introduce direction-to-direction dependencies that decrease available parallelism. This can be addressed either by iterating on the reflected angular fluxes or by accepting reduced parallelism. The best choice will be problem-dependent. Curvilinear coordinates introduce a different kind of direction-to-direction dependency, again reducing

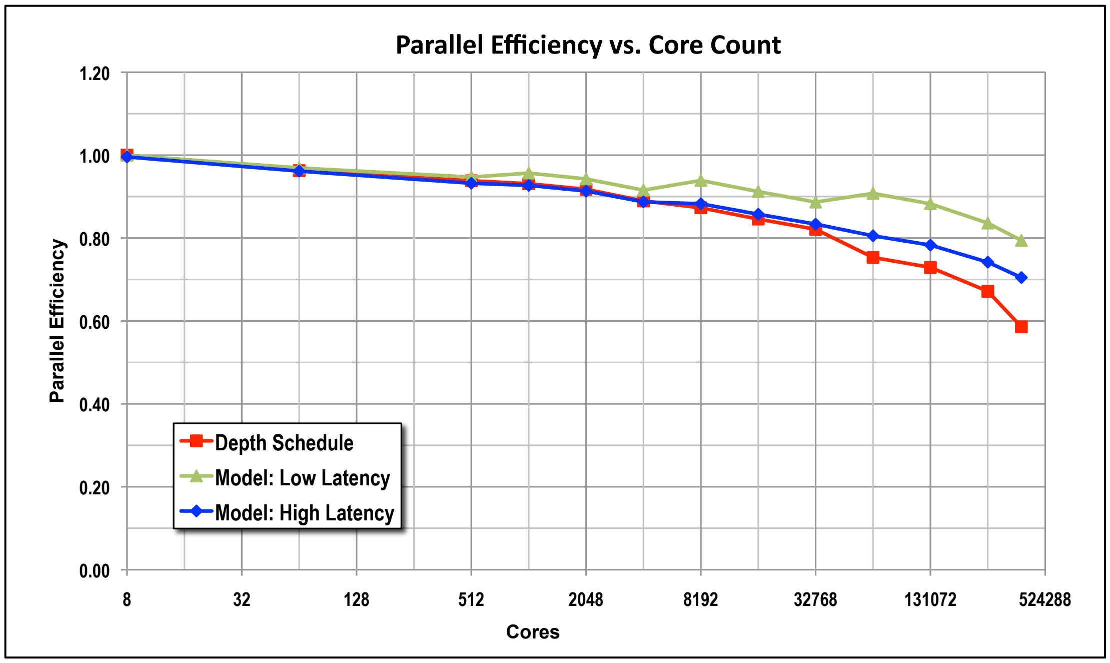

**Figure 6.** Weak scaling performance with STAPL.

available parallelism and ultimately making sweeps somewhat less efficient than in Cartesian coordinates. We are working to quantify this.

For sweeps to maintain high efficiency at processor counts of today's and tomorrow's architecture they will need to employ hierarchical parallelism. For example, on a machine with $O(10^5)$ nodes and the ability to execute $O(10^2)$ parallel threads per node, we might decompose a large problem into one spatial subdomains per node, aggregate relatively large tasks for sweeping these subdomains in parallel, and then execute each task in parallel within each node. This "inner" parallelism could be over the $A_m$ directions and $A_g$ groups associated with the task, for example. We expect to report on this in a future communication.

## ACKNOWLEDGEMENTS

Part of this work was funded under a collaborative research contract from Lawrence Livermore National Security, LLC. Part of this work was performed under the auspices of the Center for Radiative Shock Hydrodynamics at the University of Michigan, which is funded by the DOE NNSA ASC Predictive Science Academic Alliances Program. Part of this work was funded under a collaborative research contract from the Center for Exascale Simulation of Advanced Reactors (CESAR), a DOE ASCR project. This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

## REFERENCES

[1] M. L. Adams and E. W. Larsen, "Fast iterative methods for discrete-ordinates particle transport calculations," *Prog. Nucl. Energy*, **40**, No. 1, pp. 3-159, (2002).

[2] T. M. Evans, A. S. Stafford, R. N. Slaybaugh, and K. T. Clarno. Denovo: A New Three-Dimensional Parallel Discrete Ordinates Code in Scale. *Nucl. Tech.*, **171**, pp. 171-200 (2010).

[3] R. J. Zerr and Y. Y. Azmy, "Solution of the Within-Group Multidimensional Discrete Ordinates Transport Equations on Massively Parallel Architectures," *Trans. Amer. Nucl. Soc.*, **105**, 429 (2011).

[4] R. S. Baker and K. R. Koch, "An Sn Algorithm for the Massively Parallel CM-200 Computer," *Nucl. Sci. Eng.*, **128**, p. 312 (1998).

[5] T. S. Bailey and R. D. Falgout, "Analysis Of Massively Parallel Discrete-Ordinates Transport Sweep Algorithms With Collisions," *Proc. International Conference on Mathematics, Computational Methods & Reactor Physics*, Saratoga Springs, May 3-7, CDROM (2009).

[6] S. D. Pautz *Nucl. Sci. Eng.*, **140**, 111 (2002).

[7] W. Daryl Hawkins, Timmie Smith, Michael P. Adams, Lawrence Rauchwerger, Nancy M. Amato, and Marvin L. Adams, "Efficient Massively Parallel Transport Sweeps," *Trans. Amer. Nucl. Soc.*, **107**, pp. 477-481 (2012).

[8] A. Buss, HARSHVARDAN, I. PAPADOPOULOS, O. PEARCE, T. SMITH, G. TANASE, N. THOMAS, X. Xu, M. BIANCO, N. M. AMATO, L. RAUCHWERGER, "STAPL: Standard Template Adaptive Parallel Library," *SYSTOR*, Haifa, Israel, June 4-6, 2010, ACM, pp.1–10, http://doi.acm.org/

[9] G. TANASE, A. BUSS, A. FIDEL, HARSHVARDAN, I. PAPADOPOULOS, O. PEARCE, T. SMITH, N. THOMAS, X. SU, N. MOURAD, J. VU, M. BIANCO, N. M. AMATO, L. RAUCHWERGER, "The STAPL Parallel Container Framework," *Proc. ACM SIGPLAN Symp. Prin. Prac. Par. Prog.* (PPOPP), (2011).

[10] A. BUSS, A. FIDEL, HARSHVARDAN, T. SMITH, G. TANASE, N. THOMAS, X. XU, M. BIANCO, N. M. AMATO, L. RAUCHWERGER, "The STAPL pView," *LCPC*, Houston, October 7-9, (2010).